Using AMPL/OSL

OSL is an optimization package that supports linear, quadratic, network and integer programming. This supplement to *AMPL: A Modeling Language for Mathematical Programming* summarizes all of the most important features of OSL for users of AMPL. For a more extended treatment of OSL, consult *Optimization with OSL* by Ming S. Hung, Walter O. Rom, and Allan D. Waren, also published by The Scientific Press. The complete OSL reference manual, entitled *Optimization Subroutine Library Guide and Reference*, is available as document SC23-0519-03 through offices of the IBM Corporation.

Section §1 below describes the kinds of problems to which OSL is applicable. Section §2 explains in general terms how directives can be passed from AMPL to OSL, and Section §3 describes directives for allocating memory, regulating output, and other actions that are applicable to all of OSL's optimization routines. The remaining four sections cover the different kinds of optimization routines within OSL:

§4 Simplex methods for linear programming and network optimization
§5 Interior-point methods for linear programming
§6 An extended simplex method for quadratic programming
§7 Branch-and-bound procedures for pure and mixed linear integer programming

Each of these sections briefly introduces the algorithms that OSL uses, then categorizes and explains the relevant directives.

§1 Applicability

OSL is designed to solve linear programs as described in Chapters 1–8 and 11–12 of *AMPL: A Modeling Language for Mathematical Programming*, as well as the integer linear programs described in Chapter 15. Integer programs may be pure (all integer variables) or mixed (some integer and some continuous variables); integer variables may be binary (taking values 0 and 1 only) or may have more general lower and upper bounds.

For the network linear programs described in Chapter 12, OSL also incorporates an especially fast network optimization algorithm.

OSL offers an efficient specialized algorithm for minimizing a convex quadratic function or maximizing a concave quadratic function, subject to linear constraints. There are no OSL routines for dealing with the more general class of nonlinear optimization problems discussed in Chapter 13, however. Thus AMPL will reject any attempt to apply OSL to objective functions other than linear or quadratic, or to nonlinearities in the constraints:

```
ampl: model nltransd.mod; data nltrans.dat;
ampl: option solver osl;
ampl: solve;
OSL 1.2:
Sorry, osl can't handle nonlinearities.
exit code 1
ampl:
```

This restriction applies to models that use any function of variables that AMPL identifies as ‘‘not linear’’ — even a function such as `abs` or `min` that exhibits some properties of linear functions.

AMPL does enable OSL to solve piecewise-linear programs, as described in Chapter 14, by transforming them to problems that OSL's optimization routines can handle. The transformation is to a linear program, if the following conditions are met. Any piecewise-linear term in a minimized objective must be convex, its slopes forming an increasing sequence as in this example:

```
<<-1,1,3,5; -5,-1,0,1.5,3>> x[j]
```

Any piecewise-linear term in a maximized objective must be concave, its slopes forming a decreasing sequence as in:

```
<<1,3; 1.5,0.5,0.25>> x[j]
```

Any piecewise-linear term in the constraints must be either convex and on the left-hand side of a $\leq$ constraint (or equivalently, the right-hand side of a $\geq$ constraint), or else concave and on the left-hand side of a $\geq$ constraint (or equivalently, the right-hand side of a $\leq$ constraint). In all other cases, the transformation is to a mixed-integer program. AMPL automatically performs the appropriate transformation, sends the resulting linear or mixed-integer program to OSL, and converts the solution back. The transformation has the effect of adding a variable to correspond to each linear piece; when the above conditions are not met, the added variables are subject to logical constraints (known as ‘‘special ordered sets of type 2’’) which OSL's mixed integer solver handles efficiently.

§2 Controlling OSL from AMPL

In many instances, you can successfully apply OSL by simply specifying a model and data, setting the solver option to `osl`, and typing `solve`. For larger linear programs and especially the more difficult integer programs, however, you may need to pass specific directives to OSL to obtain the desired results.

To give directives to OSL, you must first assign an appropriate character string to the AMPL option called `osl_options`. When OSL is invoked by `solve`, it breaks this string into a series of individual directives. Here is an example:

```
ampl: model steelT.mod;
ampl: data steelT.dat;

ampl: option solver osl;
ampl: option osl_options 'crash=2 dual \
ampl?    tolpinf=1.0e-9 simplex=1 \
ampl?    maxiter=100';

ampl: solve;
OSL 1.2:
crash=2
dual
tolpinf=1.0e-9
simplex=1
maxiter=100
   Dual feasibility will be maintained in EKKCRSH, if possible
   There are 7 nonzero pivots
   Switching to Devex pricing--Growth of factorization
OSL 1.2: optimal solution; objective 515033
16 simplex iterations
ampl:
```

OSL confirms each directive; it will display an error message if it encounters one that it does not recognize.

OSL directives consist of an identifier alone, or an identifier followed by an `=` sign and a value; a space may be used as a separator in place of the `=`. Unlike AMPL, OSL treats upper-case and lower-case letters as being the same.

You may store any number of concatenated directives in `osl_options`. The example above shows how to type all the directives in one long string, with the \ character used to indicate that the string continues on the next line. Alternatively, you can list several strings, which AMPL will automatically concatenate:

```
ampl: option osl_options 'crash=2 dual'
ampl?     ' tolpinf=1.0e-9 simplex=1'
ampl?     ' maxiter=100';
ampl:
```

In this form, you must take care to supply the space that goes between the directives; here we have put it before `tolpinf` and `maxiter`. If you have specified the above directives, and then want to try setting, say, `toldinf` to `1.0e-8` and changing `crash` to 1, you might think to type:

```
ampl: option osl_options 'toldinf=1.0e-8 crash=1';
```

This will replace the previous `osl_options` string, however; the other previously specified directives such as `tolpinf` and `maxiter` will revert to their default values. (OSL supplies a default value for every directive not explicitly specified; each directive's definition in the following sections includes an indication or discussion of its default.) To append new directives to `osl_options`, use this form:

```
ampl: option osl_options $osl_options
ampl?     ' toldinf=1.0e-8 crash=3';
```

A `$` in front of an option name denotes the current value of that option, so this statement appends more directives to the current directive string. As a result the string contains two directives for `crash`, but the new one overrides the earlier one.

§3 General-purpose OSL directives

The following directives are applicable to all OSL algorithms. Look here for information on expanding available memory, changing the sense of optimization, and generating additional diagnostic listings.

In these and all subsequent descriptions of directives, the letters *i* and *r* denote integer and real values, respectively.

dspace= *i* **(default 0)**

Based on the size and characteristics of your problem, OSL decides how big a memory region to allocate for the algorithm that you choose. By specifying $i > 0$, you tell OSL to increase this allocation by *i* ''double words''; a double word, equal to 8 bytes in current computer architectures, is the space required for one double-precision floating-point number.

You need to use this directive only if OSL reports an error such as

```
More space is needed for the adjacency matrix
```

or

```
There is not enough storage for a factorization of
or a copy of the matrix after 3571 elements
```

and returns to AMPL without reporting an optimal solution. There is no reliable way to predict how large *i* should be, particularly for integer programs. The best approach is to start with a conservative value such as 50000, and to keep increasing it until OSL no longer reports an error. Or, you may start with a very large value, such as 1000000, which you are fairly certain will be sufficient; but if the error message changes to something like

```
malloc(33108556) failure: ran out of memory.
```

then you have made *i* too large for your computer's resources, and you will have to try a smaller value.

If you cannot find a value of *i* that avoids one kind of error message or the other, then your computer cannot make enough memory available for OSL to solve your optimization problem. The computer may have insufficient real memory installed, or it may be unable to create sufficient virtual memory due to a lack of swap space on disk. To decide exactly what the problem is, you may have to consult an expert or documentation on the particular computer you are working with.

In some cases, a larger value of *i* may allow OSL to run more efficiently, even though a smaller value is sufficient. This is especially true for integer programming; this issue is addressed further in the discussion of directives `bb_mfile` and `bb_bfile` in Section §7 below.

`minimize`
`maximize`
`maxmin=` *r*
`objno=` *i*

When none of these directives is specified, the objective function is handled by default as follows. OSL optimizes the objective most recently specified in an AMPL `objective` command, or the first (or only) objective generated from the AMPL model if no `objective` command has been given. The objective is minimized or maximized as specified by its declaration in the model. If the model declares no objectives, or if all objectives have been inactivated by AMPL's `drop` command, then OSL is invoked with an objective of 0.0, and declares optimality as soon as it finds a solution that is feasible in the constraints.

Any explicit specification of these directives will override AMPL's default handling of the objective function. OSL will optimize the *i*th objective generated from the AMPL model. Either `minimize` or $r = 1.0$ will force OSL to minimize the objective, and either `maximize` or $r = -1.0$ will force OSL to maximize. A setting of $r = 0.0$ or $i = 0$ will cause OSL to ignore any objectives and to report optimality as soon as it finds a feasible solution.

`outlev=` *i* **`(default 1)`**

This directive regulates the volume of informational messages that you receive from OSL. (Error messages are not affected.) The default of $i = 1$ tends to limit the output to short messages that are of the greatest interest. Even these messages may be suppressed by setting $i = 0$.

A setting of $i = 3$ displays all informational messages generated by OSL. This option may be useful for troubleshooting, or for displaying the progress of a long run, but the output can be very long; you may want to direct it to a file by typing `solve >`*filename*. A more concise progress report can be requested by other directives, described in the following sections, that control the output from particular optimization routines.

Settings of $i = 2$ and $i = 4$ are the same as $i = 1$ and $i = 3$, respectively, except that the informational messages are accompanied by their IBM-standard message numbers. If you need further explanation of a message, you can look it up by its number in the *Optimization Subroutine Library Guide and Reference*.

`linelen=` *i* **`(default 80)`**

Messages from OSL are broken into lines of at most *i* characters; values in the range $60 \le i \le 150$ are accepted. If your screen or window is wide enough, a setting greater than the default of $i = 80$ will generally make the messages more readable.

`trace=` *i* **`(default 0)`**
`printcpu=` *r* **`(default 0.0)`**

When increased from their default values, these directives request additional information about the execution of routines within OSL.

When $i = 1$, a message is printed each time that OSL enters a new subroutine. When $i = 2$, the message numbers are also printed.

For $r > 0$, any OSL routine that takes more than *r* seconds generates a message indicating how much time the routine took, and the total CPU time used by OSL up to that point.

`timing=` *i* **`(default 0)`**

When increased from its default value, this directive requests a timing summary at the end of the OSL run:

```
Times (seconds):
Input =  0.21
Solve =  7.03
Output = 0.16
```

The ‘‘`Solve`’’ figure is the most appropriate for comparing the speed of different algorithmic options.

When a simple `solve` is used, $i = 1$ causes the timing summary to appear on the screen. When `solve >`*filename* is employed to redirect OSL output to a file, $i = 1$ sends the summary to the file, $i = 2$ leaves it on the screen, and $i = 3$ writes it to both the file and the screen.

§4 Using OSL's simplex algorithms for linear programming

The simplex method is the best known and most widely used procedure for solving linear programs (or LPs); it is the method first introduced in most mathematical programming textbooks. OSL offers several distinct algorithms based on the simplex method, including algorithms that are applicable to any linear program and algorithms that are specialized to pure network flow LPs. Optional settings for each algorithm permit it to be tuned to the characteristics of particular problems. OSL normally chooses an algorithm and its settings for you, taking the characteristics of your problem into account, but you can override its choice by use of the directives described below.

We begin this section by introducing directives that determine how a linear program will be set up, and which algorithm will solve it. We then present directives that apply specifically to the general-purpose simplex algorithms and to the network simplex algorithms. We conclude with directives that are useful in starting and stopping simplex algorithms and in observing their progress.

Directives for setting up a problem

By default, the linear program set up and solved by OSL is essentially the same as the linear program that was generated by AMPL. By setting a few directives, however, you can instruct OSL to dualize, scale or simplify your LP before it is solved. These directives can be used in any combination; their discussion below indicates some of the most common situations in which they are useful.

OSL reverses the effects of any transformations before it sends back any optimal values. Thus the solution that you view in AMPL will be optimal for your original linear program, regardless of how you use these directives.

`primal` **(default)**
`dual`

Every linear program has an equivalent ‘‘opposite’’ linear program; the original is customarily referred to as the primal LP, and the opposite as the dual. For each variable and each constraint in the primal there are a corresponding constraint and variable, respectively, in the dual. The `primal` or `dual` directive instructs OSL to set up the primal or the dual formulation, with the primal being the default.

Either the primal or the dual formulation may solve faster, depending on the structure of the problem being solved and on the algorithm being used. The dual is most likely to be advantageous when the number of primal constraints is much larger than the number of variables, because then the dual requires a much smaller equation system to be solved at each iteration.

`scale=`*i* **(default 0)**

When *i* is set to 1, OSL attempts to bring the coefficient values of the linear program closer together by scaling the constraints and variables — or, equivalently, by scaling the rows and columns of the constraint matrix. Each column is divided by the geometric mean of its largest and smallest nonzero magnitudes (the square root of their product); a similar scaling is then applied to each row. This cycle is repeated up to 6 times, depending on its effectiveness. Finally, each column is scaled once more so that its largest magnitude is 1.0.

Simplex algorithms may run faster and more stably on the scaled problem, particularly if coefficients vary by many orders of magnitude in the original. On the other hand, scaling is best avoided when the coefficients have small integral values, such as for network flow problems.

When *i* is left at its default value of 0, no scaling is applied.

`simplify=` *i* **(default –1)**

When *i* is changed from its default of –1 to one of the following nonnegative values, OSL attempts to apply various transformations that reduce the size of the linear program without changing its optimal solution.

When $i = 0$, OSL applies several simple and fast transformations. Constraints that involve only one non-fixed variable are removed; either the variable is fixed and also dropped (for an equality constraint) or a simple bound on the variable is recorded (for an inequality). Each inequality constraint is subjected to a simple test to determine if there exists any setting of the variables (within their bounds) that can violate it; if not, it is dropped as nonconstraining. Further iterative tests attempt to tighten the bounds on variables, possibly causing additional variables to be fixed, and additional constraints to be dropped. Since AMPL's presolve phase (as described in Section 10.2 of the AMPL book) also performs these transformations, this setting makes a difference only when you have set the `presolve` option to 0.

When $i = 1$, OSL performs the simple transformations described above and also eliminates by substitution all two-variable constraints of the form $x - y = b$. Each such elimination removes both the constraint and one of the variables.

When $i = 2$, OSL performs the simple transformations described above and also eliminates by substitution all constraints of the form $x - \Sigma_i y_i = 0$ where x and the y_i are nonnegative, and x appears in at most one other constraint. Each such elimination removes both the constraint and the variable x, without increasing the number of nonzero coefficients in the linear program. This simplification is especially useful for network flow LPs in which some nodes have only one incoming or one outgoing arc.

When $i = 3$, all of the above transformations are applied.

Directives for algorithm selection

OSL normally applies a built-in criterion to choose either the general-purpose primal or dual simplex algorithm, or the faster network primal simplex algorithm. The following discussion describes directives for specifying an algorithm explicitly, and suggests some situations in which you should consider experimenting with alternatives.

`simplex=` *i* **(default 0)**

Specifying $i = 1$ or $i = 2$ instructs OSL to apply the primal or the dual variant of the simplex method, respectively. The default of $i = 0$ leaves the choice to an internal OSL criterion based on problem characteristics; OSL prints a message to let you know which variant it has chosen.

The primal and dual simplex algorithms solve linear equation systems of the same kind and dimension, but employ opposite strategies in constructing a path to the optimum. Either can be applied regardless of whether the primal or the dual LP is set up as explained above; in general all four combinations of primal/dual setup and primal/dual algorithm perform differently.

OSL's choice of simplex variant usually offers good performance. Consider switching to the alternative variant if the number of iterations is unexpectedly high (more than a small multiple of

the number of constraints) or if you receive warning messages reporting numerical difficulties in the computations.

`netalg=` *i* **`(default 1)`**

When left at its default of $i = 1$, this directive causes OSL to switch to a network primal simplex algorithm whenever the linear program received from AMPL can be identified as a pure network LP. When $i = 2$, OSL switches instead to a network dual simplex algorithm, which performs generally the same work per iteration but takes a different kind of solution path to the optimum.

Network simplex algorithms run much faster than their general-purpose counterparts, because they can take advantage of the LP's network structure to streamline the simplex computations at each iteration. Speed-ups of 10 to 100 times are common, and the advantage tends to become greater as problem size increases. Thus the use of a network simplex algorithm is always the preferred option when available. For purposes of comparison, you can force OSL to use a general-purpose simplex algorithm by specifying $i = 0$.

For an LP to be solvable by OSL's network simplex algorithms, each variable must have at most one coefficient of $+1$ and one coefficient of -1 in the constraints (aside from constraints that express upper or lower bounds on the variable), and must have coefficients of zero elsewhere. For OSL's network dual simplex algorithm, in the usual case where all variables have finite lower bounds, the coefficients in a minimized objective must all be ≥ 0, and in a maximized objective must all be ≤ 0; an inappropriate application of this variant will be rejected with a message that ''the initial point must be dual feasible.''

You can ensure that your linear program will be recognized as a pure network flow LP by specifying all variables and constraints in terms of AMPL's `node` and `arc` declarations, using only the elementary forms of the `from` and `to` phrases that do not specify loss or conversion factors. If you use the `var` and `subject to` declarations, however, then AMPL will perform some tests to see if it can detect a pure network structure. AMPL's presolve phase takes care of any constraints that merely bound or fix variables. AMPL's OSL interface then checks whether the remaining constraint coefficients are all $+1$ or -1, with at most two in each column; if so, a simple algorithm is applied to determine whether, through scaling of selected constraints by -1, the matrix can be made to have at most one $+1$ and one -1 in each column. See Chapter 11 of the AMPL book for discussion and examples of pure network LP formulations.

Directives for the general-purpose simplex algorithms

The following directives can be used to reset key strategies and parameters of OSL's general-purpose primal and dual simplex algorithms. The default values of these directives often work quite well, and should be tried first. If the default performance does not meet your application's needs, then you can experiment with other settings; some suggestions appear in the discussion below. Before proceeding to the details, we summarize the main simplex terms and properties, for which further explanation can be found in any linear programming textbook.

Simplex algorithms maintain a subset of *basic variables* (or, a *basis*) equal in size to the number of constraints. A *basic solution* is obtained by solving for the basic variables, after each of the nonbasic variables has been fixed at one or the other of its bounds. Each iteration of the algorithm picks a new basic variable from among the nonbasic ones, steps to a new basic solution, and drops some basic variable at a bound.

A basic solution is *feasible* if it satisfies all of the constraints, and *optimal* if it also gives the best possible value of the objective. The version of the simplex algorithm implemented in OSL iterates toward feasibility and optimality simultaneously, rather than iterating toward feasibility first and then iterating toward optimality while maintaining feasibility. As a result, no measure of infeasibility is strictly decreasing from one iteration to the next, and feasible iterates can be followed by infeasible ones.

The coefficients of the variables form a *constraint matrix*, and the coefficients of the basic variables form a nonsingular square submatrix called the *basis matrix.* At each iteration, the sim-

plex algorithm must solve certain linear systems involving the basis matrix. For this purpose OSL maintains a *factorization* of the basis matrix, which is updated at most iterations, and is occasionally recomputed.

The *density* of a matrix is the proportion of its elements that are not zero. The constraint matrix, basis matrix and factorization are said to be relatively *sparse* or *dense* according to whether their densities are low or high. Most linear programs of practical interest have many zeros in all the relevant matrices, and the larger LPs tend also to be the sparser.

The amount of memory required by OSL's simplex routines grows with the size of the linear program, which is a function of the numbers of variables and constraints and the sparsity of the coefficient matrix. The factorization of the basis matrix also requires an allocation of memory; the amount is problem-specific, depending on the sparsity of the factorization.

`crash=` *i* **(default 0)**

This directive governs OSL's choice of a basis from which to start the simplex algorithm, except when the basis is read from a file as specified by the directive `startbasis` described below. The default of $i = 0$ chooses an ''obvious'' initial basis consisting of a slack variable on each inequality constraint and an extra artificial variable on each equality constraint. (All artificial variables must be zero in any feasible solution.) Other settings employ heuristics to construct initial bases that have fewer artificial variables and that tend to be fewer iterations away from the optimum. The effectiveness of crash heuristics in any particular application can be established only by experiments, but one of the following is likely to be advantageous whenever the basis of slacks and artificials is not feasible. Experience suggests that $i = 2$ gives the best results most often, followed by $i = 1$.

For $i = 1$, OSL simply tries to find a nonsingular starting basis that has few artificial variables.

For $i = 2$, OSL restricts the initial basis to variables that have a coefficient of 0 in the objective. As a result the number of artificial variables may be greater than when $i = 1$, but the initial basis is feasible for the dual LP, and may provide a better start for the dual simplex algorithm.

For $i = 3$, the initial basis is guaranteed to have a sum of infeasibilities that is no larger than the sum of infeasibilities for the $i = 0$ option. Although the number of artificial variables may be greater than for $i = 1$, the total iterations required may be less.

The setting $i = 4$ combines the features of $i = 2$ and $i = 3$.

`simplexinit=` *i* **(default 0)**

When directive `crash` is left at its default of 0 and directive `startbasis` is not set, the `simplexinit` directive governs how OSL's simplex algorithms make use of the current values of the primal variables supplied by AMPL. At the default setting of $i = 0$, these values are ignored.

When $i = 1$, each nonbasic variable is moved to the bound nearest to its current value.

When $i = 2$, for the primal simplex algorithm only, every nonbasic variable is left at its current value, even if its value is not at a bound. (OSL employs a simple extension to the standard primal algorithm that allows a nonbasic variable not at bound to be considered for entry to the basis by either increasing or decreasing from its current value.) This setting forces the `pricing` directive described below to its default value of 2.

`pricing=` *i* **(default 2)**

Any variable is eligible to enter the basis if its *reduced cost*, which may be computed at the beginning of an iteration, is negative when minimizing or positive when maximizing. OSL offers two principal ''pricing'' strategies for deciding which eligible variable to select at each iteration. When $i = 1$, all iterations use a ''devex'' pricing strategy that is usually effective in avoiding excessively large numbers of iterations. At the default setting of $i = 2$, OSL initially uses a faster ''random'' pricing strategy, switching later to devex.

To further tune the pricing strategy, you may employ two additional directives described next. The `devexmode` directive chooses among several devex routines, and the `fastits` directive influences the criterion for switching from random to devex pricing.

devexmode= *i* **(default 1)**

At the default setting of $i = 1$, OSL uses an ''approximate'' devex pricing strategy that offers a reasonable tradeoff between number of iterations and cost per iteration. In some cases, however, a better tradeoff may be obtained by choosing one of the following alternatives.

When $i = 0$, devex pricing is switched off. Cost per iteration is somewhat less as a result, but the number of iterations can increase considerably.

When $i = 2$, OSL employs a strategy that increases the cost per iteration but offers a potential for a smaller number of iterations. (In technical terms, the strategy is ''exact'' devex for the primal simplex algorithm, and ''steepest edge with inaccurate initial norms'' for the dual simplex.) For the primal simplex, $i = -2$ requests approximate devex when the solution is infeasible, and exact devex when it is feasible.

When $i = 3$, OSL employs an ''exact steepest edge'' strategy that further increases the cost per iteration, but offers the greatest potential for reduction in the number of iterations. For the primal simplex, $i = -3$ requests approximate devex when the solution is infeasible, and exact steepest edge when it is feasible.

Although performance varies widely from one application to another, the best results have most often been given by setting *i* to 1 or -2 for primal simplex, and to 3 for dual simplex.

fastits= *i* **(default 0)**

When the `pricing` directive is left at its default of 2, this directive determines when the switch is made from random to devex pricing. OSL's random strategy attempts to select a variable cheaply, by examining only a random subset of the reduced costs from the previous iteration. The selected variable's current reduced cost is then computed, and if it has the right sign the variable is brought into the basis. If it fails to have the right sign, then all reduced costs are recomputed and an alternative variable is selected.

The default of $i = 0$ decides when to switch from random to devex by use of an internal heuristic, based on the density of the basis factorization and the success rate of random pricing. (A dense factorization implies that pricing will represent a relatively small part of the cost of an iteration, in which case it makes sense to use a more costly but more accurate strategy such as devex.)

When *i* is positive, OSL carries out random pricing as above for the first *i* iterations, then switches to random pricing with reduced costs recomputed at every iteration. At the next refactorization of the basis, the strategy is switched to devex.

maxfactor= *i* **(default 100+***m***/100, where** *m* **= number of constraints)**

OSL recomputes the basis factorization at least once every *i* iterations. The performance of the simplex algorithm is usually fairly insensitive to this setting.

pweight= r_1 **(default 0.1)**
dweight= r_2 **(default 0.1)**
changeweight= r_3 **(default 0.5)**

OSL's simplex algorithms employ an objective function of the form

$$\alpha \text{ (true objective)} \pm \text{(sum of infeasibilities)}$$

where the sum of infeasibilities is added when minimizing and subtracted when maximizing. For all sufficiently small α, a solution is optimal for this composite objective if and only if it is optimal for the true objective; but the composite objective has the advantage of permitting the simplex algorithm to iterate toward feasibility and optimality at the same time.

Initially, α is set to the positive value given by r_1 (in the primal simplex algorithm) or r_2 (in the dual simplex algorithm). The default of 0.1 usually works well, though a smaller value may be appropriate when the initial basis is feasible.

As the algorithm proceeds, an internal heuristic decreases α at each iteration, by a large amount if the sum of infeasibilities increases, and a small amount if progress toward feasibility is slow.

The setting of r_3 governs the rate of decrease employed by the heuristic, with the default of 0.5 usually giving a reasonable change. A larger value ≤ 1.0 gives a greater decrease, while a smaller value > 0 gives a lesser decrease.

tolpinf=r_1 **(default 1.0e–8)**
toldinf=r_2 **(default 1.0e–7)**

A solution to a linear program is considered to be feasible if no bound or constraint is violated by more than r_1. A solution is considered to satisfy the optimality conditions (that is, to be feasible in the dual constraints) if no reduced cost is less than $-r_2$ when minimizing, or greater than r_2 when maximizing.

The default value of r_1 is usually adequate. It may need to be increased, however, if the constraints can only be satisfied to a lesser precision; in such a case OSL will report ''no feasible solution'' but all of the infeasibilities will be very small. A modest increase in r_1, say to 1.0e–7 or 1.0e–6, can also give the algorithm added flexibility and reduce the number of iterations required; but too great an increase will make the algorithm unreliable.

The default value of r_2 is also usually adequate. A decrease (say, to 1.0e–8) may cause the algorithm to find a slightly better objective value, while an increase (say, to 1.0e–6) may cause optimality to be declared after fewer iterations.

Directives for the network simplex algorithms

Because OSL's network simplex algorithms are very specialized versions of the regular simplex algorithms, they do not respond to the same directives. The following may be used to influence the network simplex strategies for constructing an initial basis and for selecting an entering variable.

netinit=*i* **(default 0)**

At its default value of 0, this directive tells the network simplex algorithm to start from a basis of artificial variables. All artificial variables must be zero in a feasible solution, and hence they can be ignored when an optimum is returned.

A setting of $i = 1$ directs OSL to develop an initial basis from the current values of the variables that AMPL supplies.

netprice=*i* **(default 0)**
netsamp=*r* **(default 0.05)**

Since determination of reduced costs for the nonbasic variables tends to be the most expensive part of a network simplex algorithm, OSL computes only a sample of the reduced costs at each iteration. Under the default of $i = 0$, an internal OSL heuristic is used to select a sample size and to adjust it as the algorithm proceeds. When $i = 1$, the sample size is set at r times the number of variables, for all iterations.

Directives for starting and stopping

OSL always sends a solution back to AMPL, where you can use commands such as `display` to view the results. The following directives let you also save and restore simplex bases, and let you stop the simplex method after a specified number of iterations.

endbasis=f_1
startbasis=f_2

If the `endbasis` directive is specified, OSL writes a record of the final simplex basis to the file named f_1. This file, in the standard MPS basis format (see *Optimization with OSL* or the *Optimization Subroutine Library Guide and Reference*), contains a list of all basic variables, as well as an indication of the value at which each nonbasic variable lies. Normally the recorded basis is optimal, but it may be otherwise if an optimum does not exist or could not be found by the chosen algorithm, or if the iterations were terminated prematurely by the `maxiter` directive described below.

By default, OSL employs a heuristic procedure to determine a starting basis for the simplex algorithm, as indicated in the discussion of the `crash` directive above. If the `startbasis` directive is specified, the initial basis is instead read from the file f_2, which must also be in the standard MPS basis format. This basis then determines the initial solution, regardless of any initial values that the variables might have in AMPL.

This feature can be useful for quickly solving a series of linear programs that differ only in the data. Before the first one is solved, `endbasis` is set to the name of a temporary file:

```
ampl: model steelT3.mod; data steelT3.dat;
ampl: option solver osl;
ampl: option osl_options 'endbasis=/tmp/steelT3.bas';
ampl: solve;
OSL 1.2
 The primal algorithm has been chosen.
OSL 1.2: optimal solution; objective 514521.7143
34 simplex iterations
```

Then a data value is changed, and `startbasis` is set to the same filename so that the previously optimal basis is used as a start:

```
ampl: let avail[1] := 32;
ampl: option osl_options 'startbasis=/tmp/steelT3.bas';
ampl: solve;
OSL 1.2
 The dual algorithm has been chosen.
OSL 1.2: optimal solution; objective 493648.7143
1 simplex iterations
```

Only 1 iteration is necessary to step to an adjacent basis that is optimal for the modified linear program. Although OSL will try to make use of any previously saved basis specified by `startbasis`, this approach should be relied upon only when the change to the data does not affect the numbers of variables and constraints. (These numbers can be reduced by simplifications carried out in AMPL's presolve phase, even when it would seem that a change in the data should have no effect. If the number of iterations is unexpectedly high, use the AMPL command `option show_stats 1` to see what presolve has done, or use `option presolve 0` to turn presolve off.)

Another use for this feature is to restart OSL, possibly with some directives changed, after it has reached an iteration limit short of optimality due to the `maxiter` directive described next.

`maxiter=` *i* **(default 9999999)**

OSL stops after *i* simplex iterations and returns its current solution, whether or not it has determined that the solution is optimal.

Directives for controlling output

If your linear program will take a long time to solve, you may want to use the following directive to monitor its progress. You may also request more complete diagnostic information by increasing the setting of the `outlev` directive described previously in Section §3.

`logfreq=` *i* **(default 0)**

The default of $i = 0$ produces a minimal few lines of output from OSL, summarizing the results of the run.

When $i > 0$, a ''log line'' recording the iteration number, current objective value and other information is displayed every *i* iterations, and at every refactorization of the basis, change in the weight on the true objective (see `changeweight`) and switch from random to devex pricing (see `pricing`). Some log lines may be more readable if you increase the directive `linelen` from its default value of 80 to 90 or 100.

§5 Using OSL's interior-point algorithms for linear programming

OSL incorporates an alternative class of algorithms for linear programming, which have been found to be faster than simplex algorithms on many types of linear programs. These are commonly known as *interior-point* algorithms, and are also called *barrier* algorithms in much of the OSL documentation. Barrier algorithms require a number of iterations that increases very slowly with problem size, so that their advantage tends to grow as LPs get larger. They are especially likely to be advantageous when the number of variables is much larger than the number of constraints, or when the simplex algorithms are stalled by many degenerate iterations at which no improvement in the objective function is achieved.

Barrier algorithms are more likely to encounter difficulty when there are many ''free'' variables (not bounded above or below), and when there are ''dense'' columns in the constraint matrix — as a result of some variables appearing in a large number of constraints. They also tend to be less advantageous relative to simplex algorithms when the number of constraints is large relative to the number of variables.

We begin this section with the directives for setting up a problem and selecting an algorithm. The directives for influencing an algorithm are then presented in two groups: those that pertain to the solution of linear equation systems, and those that guide the sequence of iterates. Directives to limit the number of iterations and to obtain more detailed output are discussed at the end of this section.

Directives for setting up a problem

The `primal`, `dual`, `scale` and `simplify` directives work in the same way as for the simplex algorithms. It is not advisable, however, to use `scale` with the primal-dual barrier algorithms (see the `barrier` directive below).

Directives for algorithm selection

By default, OSL uses only simplex algorithms. One or both of the following directives must be set to request a barrier algorithm, or a barrier algorithm followed by a simplex algorithm.

`barrier=` i **(default –1)**

A setting of $i \geq 0$ overrides the `simplex` and `netalg` directives, and selects a barrier algorithm instead. Positive values specify particular variants as follows:

$i = 1$ primal barrier
$i = 2$ primal-dual barrier
$i = 3$ primal-dual barrier with predictor-corrector steps

The primal-dual algorithm with predictor-corrector steps is usually the fastest and the most stable of these three, and should be considered the variant of choice.

Specifying $i = 0$ lets OSL choose an appropriate barrier algorithm, while the default of $i = -1$ results in a simplex method being used.

`bs_switch=` i_1 **(default 0)**
`possbasis=` i_2 **(default 1)**

When i_1 is left at its default value of 0, the barrier algorithm iterates to completion and the resulting solution is returned to AMPL. When the optimal solution is not unique, the solution returned by a barrier algorithm is a kind of average of all possible optimal solutions. In particular, any variable that lies strictly between its bounds in *some* optimal solution is likely to lie between its bounds in the optimal solution returned by the barrier algorithm.

Positive values of i_1 specify that OSL should switch to a simplex algorithm before returning a solution, as follows:

i_1=1 switch if numerical instabilities are detected
i_1=2 also switch when the barrier algorithm is finished
i_1=3 also switch immediately if OSL determines that a simplex algorithm would be more appropriate

The simplex algorithm finds an ''extreme'' solution that tends to have more variables at their bounds. Thus the solution returned when $i_1 = 2$ may be quite different from the one returned when $i_1 = 0$.

The setting of i_2 regulates information passed along when a switch to the simplex method is requested. When i_2 is at its default value of 1, certain variables are marked as ''potentially basic'' at the end of the barrier algorithm; such information often helps the simplex algorithm find a basic solution faster. This feature may be suppressed by setting i_2 to 0.

Directives for the equation-solving routines

At each iteration, the barrier algorithms solve a certain system of linear equations, or several such systems that have the same coefficients but different right-hand sides. The coefficient matrix has the symmetric form ADA^T, where A is the constraint matrix of the linear program and D is a diagonal matrix that depends on the current values of the variables.

Most of the computational time in the barrier algorithms is devoted to solving these equations. OSL's equation-solving routines perform Gaussian elimination to determine a Cholesky factorization that should be very efficient for typical linear programs. Nevertheless, in some cases these routines may be accelerated by resetting one or more of the following directives.

`adjactype=` i_1 **(default 1)**
`formntype=` i_2 **(default 0)**

These directives control a tradeoff between time and space in the formation of the coefficient matrix ADA^T. The `adjactype` directive determines how OSL constructs a map (the ''adjacency graph'') of the positions of nonzero elements in ADA^T; this procedure is performed only once at the start of the algorithm. The `formntype` directive determines how OSL computes the numerical values of the elements of ADA^T at each iteration. In general terms, settings of 0 use routines based on outer products of columns of A, while settings of 1 use routines based on inner products of rows of A.

The default settings give the fastest performance, but require the greatest amount of computer memory.

When $i_1 = i_2 = 0$, the formation of the adjacency graph is somewhat slower, but OSL can save the memory that would otherwise be required for a row-wise copy of A.

When $i_1 = i_2 = 1$, the formation of ADA^T at each iteration tends to be slower, particularly on computers that can perform some kind of vector or parallel processing; but the requirement for memory is the lowest. OSL automatically switches i_2 to 1 if there is not enough memory available to use the default setting.

`densecol=` i **(default 99999)**

OSL operates on the assumption that A is a ''sparse'' matrix whose elements are mostly zeros, and that ADA^T is consequently also sparse. If even one column of A has many nonzeros, however, then ADA^T may be much denser than A and the barrier algorithm may become impractically slow. To circumvent this problem, OSL incorporates optional equation-solving routines that handle ''dense'' columns of A in a special way. (In technical terms, the non-dense part of A is used as a preconditioner to solve the linear systems by a conjugate gradient method.)

Any column that has more than i nonzero elements is treated as dense by OSL. Thus at the default setting, no dense columns are identified. For linear programs that are known to have dense columns, a smaller positive value of i may improve performance markedly; some experimentation may be necessary to determine the proper setting, but a value of 20 is often a good start. (To see

the maximum nonzeros that OSL finds in any column, and how many columns OSL regards as dense, set the `outlev` directive to 3.)

densethr= *r* **(default 0.7)**

As elimination on ADA^T proceeds, additional nonzero elements are created in the uneliminated part. Thus OSL uses sparse elimination techniques at the outset, but switches to a dense elimination routine — storing all zeros explicitly — when it encounters a column of the uneliminated matrix whose proportion of nonzeros is greater than *r*.

The computational time in barrier algorithms is often insensitive to the value of *r*, but experimentation may reveal a noticeable difference for some problems. A setting of $r = 0.0$ requests that dense processing be used from the beginning, while $r = 1.0$ suppresses the dense elimination routines.

droprowct= *i* **(default 1)**
cholabstol= r_1 **(default 1.0e–15)**
choltinytol= r_2 **(default 1.0e–18)**
pertdiag= r_3 **(default 1.0e–12)**

As OSL carries out Gaussian elimination on ADA^T, it may find some constraints that are probably redundant; that is, they are likely to be implied by other constraints. When *i* is at its default value of 1, any such constraints are dropped for the remainder of the algorithm.

If OSL unexpectedly reports an unbounded solution, it may have dropped constraints that were not actually redundant. For $2 \le i \le 16$, a constraint is dropped only if it is detected as redundant at *i* consecutive iterations. For $i > 16$, constraints are never dropped.

In technical terms, a constraint is dropped when, during the Gaussian elimination of ADA^T, the corresponding ''pivot element'' on the diagonal is too small. The handling of small pivot elements is regulated by the settings of $r_1 \ge r_2$. Any element between r_1 and r_2 is arbitrarily increased to r_1, while for any element smaller than r_2 the associated constraint is dropped for the current iteration (and possibly for all later iterations depending on *i*).

To discourage spurious dropping of constraints, OSL perturbs the diagonal matrix *D* by adding r_3 to each diagonal element.

nullcheck= i_1 **(default 0)**
maxprojns= i_2 **(default 3)**
projtol= *r* **(default 1.0e–6)**

When i_1 is left at its default value, OSL checks certain solutions to key equation systems in the barrier algorithm. If the norm of the discrepancy between the left-hand and right-hand sides is less than *r* times the norm of the solution, then the solution is accepted. Otherwise, OSL applies a refinement procedure to the solution and checks again; as many as $i_2 - 1$ refinements may be attempted.

Checking and refinement may be suppressed by setting i_1 to a negative value or i_2 to 1. This can reduce the solution time somewhat for problems that are numerically well behaved in the barrier algorithms.

Directives for computing the iterates

Once the appropriate equations have been solved to determine a step direction, a barrier algorithm determines the maximum *step length* that keeps certain variables within their bounds (the details depending on the variant chosen). The actual step length is then taken to be some amount less than the maximum — so as to keep the step ''in the interior'' — and a new iterate is determined.

These algorithms also maintain a *barrier parameter* μ that determines, roughly, how much the next iterate will be forced toward the center of the feasible region. A larger $\mu > 0$ implies more centering, which is desirable at early iterations; the value of μ is gradually reduced toward zero as the iterates approach optimality.

The following directives influence, in various ways, the computation and treatment of the iterates in the barrier algorithms.

tolpinf=r_1 **(default 1.0e-8)**
toldinf=r_2 **(default 1.0e-7)**

A solution to a linear program is considered to be feasible if no bound or constraint is violated by more than r_1. A solution is considered to satisfy the optimality conditions — that is, to be feasible in the dual constraints — if none is violated by more than r_2.

These directives work the same as for the simplex algorithms, and are discussed further in §4 above.

fixvar1=r_1 **(default 1.0e-7)**
fixvar2=r_2 **(default 1.0e-8)**

For some linear programs, a feasible solution can be formed only when certain variables lie at their upper or lower bounds. This property can slow the progress of interior-point algorithms, which keep variables strictly between their bounds. Thus, after each step, any variable that has moved to within r_1 of a bound (when the current iterate is infeasible) or to within r_2 of a bound (when the current iterate is feasible) is fixed at that bound for all remaining iterations.

Moving these directives from their default settings may result in a change to the number of iterations required. Generally it is best to set $r_1 > r_2$, and to keep r_2 small.

mulinfac=r **(default 0.0)**

When there is more than one optimal solution, barrier algorithms tend to converge to the ‘‘center’’ of the optimal region. Consequently, if this region is not bounded, certain variables in the solution will tend toward ∞ as the iterates approach optimality.

This directive attempts to discourage such behavior by adding $r\mu$ to each variable's coefficient in the objective function, where μ is the barrier parameter described above. Normally r should be left at its default of 0.0, which has no effect. If you find some variables at unexpectedly large values in the optimal solution, however, then a small positive r may give a more acceptable result; a setting of .001 or .01 is a good place to start.

Unboundedness of the optimal region is often the result of a ‘‘free variable’’ that has been transformed to the difference of two nonnegative variables. The barrier algorithms will perform better if all free variables are declared explicitly in the AMPL model; the `var` declaration for a free variable is the same as for any other variable, except that no lower or upper bound is specified.

pdstepmult=r_1 **(default .99995)**
pdgaptol=r_2 **(default 1.0e-7)**

These are the key parameters influencing the behavior of the primal-dual barrier algorithms.

The actual step length at each iteration is r_1 times the maximum step length that is within the bounds of the variables. Normally this number should be kept slightly less than 1, so that the step is nearly as large as possible yet remains within the interior. OSL allows r_1 to be set as high as .999999, which may reduce the number of iterations marginally. A somewhat lower value of r_1 may be helpful if the algorithm appears to be having numerical difficulties.

The algorithm stops when the current primal and dual iterates satisfy the `tolpinf` and `toldinf` feasibility tolerances, and the relative difference in the objective values of the primal and dual LPs is less than r_2. Writing z_p and z_d for the objective values attained by the primal and dual iterates, the relative difference is defined as

$$(z_p - z_d)/(1+|z_d|).$$

Upon return from a primal-dual barrier algorithm, AMPL reports both z_p and z_d, which necessarily differ by at most
$r_2(1+|z_d|)$. The setting of r_2 can safely be increased or decreased by a few factors of 10 from its default, with only a minor effect on the number of iterations.

objweight= r_1 **(default 0.1)**
muinit= r_2 **(default 0.1)**
rgfactor= r_3 **(default 0.1)**
stepmult= r_4 **(default 0.99)**
mufactor= r_5 **(default 0.1)**
mulimit= r_6 **(default 1.0e–8)**
rglimit= r_7 **(default 0.0)**

These are the key parameters influencing the behavior of the primal barrier algorithm.

This variant proceeds in two phases, the first to seek a feasible solution and the second to seek an optimal one. In the first phase, r_1 is the weight given to the true objective function. Smaller values cause a feasible solution to be attained faster, while larger values tend to yield an initial feasible solution that is nearer to optimal.

At the beginning of each phase, the initial value of the barrier parameter μ is determined from r_2 by a scaling routine internal to OSL. If the objective function will get very large, it may help to set r_2 to a larger value such as 1000. At the first iteration, OSL also computes the norm of a *reduced gradient* vector as an indicator of the rate at which the objective is being decreased. Typically, the reduced gradient norm gets smaller as the algorithm proceeds with a given value of μ. A target for reduction of the norm is set at r_3 times the initial norm.

At each iteration, the step length is set to r_4 times the maximum length that is within the bounds of the variables. Normally this number should be kept slightly less than 1, so that the step is nearly as large as possible yet remains within the interior. OSL allows r_4 to be set as high as .99999, which may reduce the number of iterations marginally. A somewhat lower value of r_4 may be helpful if the algorithm appears to be having numerical difficulties.

A reduced gradient norm is also computed at each iteration. When it drops below the previously set target, μ is reduced by a factor of r_5. The target is then reset to r_3 times the current norm.

The current phase terminates when μ has been reduced below r_6 (scaled in the same way as r_2) and the reduced gradient norm has again dropped below the target; a smaller value of r_6 may yield a more accurate solution. The current phase will also terminate if the reduced gradient norm is found to be less than r_7 at any iteration; this criterion has an effect only if r_7 is increased from its default of 0.0 to some small positive value.

Directives for starting and stopping

To operate reliably and efficiently, OSL's implementations of the barrier algorithms must choose a starting point in a certain way. Thus the options of ''saving the basis'' or ''restarting from a previous optimum'' for the simplex algorithms have no counterparts for the barrier algorithms. The following option does permit you to stop a barrier algorithm after a specified number of iterations.

maxiterb= *i* **(default 100)**

OSL stops after *i* iterations of the barrier algorithm and returns the current solution, whether or not it has determined that the solution is optimal. Barrier algorithms seldom require more than the default of 100 iterations; if OSL reaches this limit, then the variant you have chosen is probably not working effectively, and you should choose a different barrier variant or switch to a simplex algorithm.

Directives for controlling output

If your linear program will take a long time to solve, you may want to use the following directive to monitor its progress. You may also request more complete diagnostic information by increasing the setting of the `outlev` directive described previously in Section §3.

logfreqb= *i* **(default 0)**

The default of $i = 0$ produces a minimal few lines of output from OSL, summarizing the results of the run.

When $i > 0$, a ''log line'' recording the iteration number, current objective value and other information is displayed every i iterations. For the primal-dual variants of the barrier algorithm, log lines give the current objective values of both the primal and dual iterates, as well as the gap between them, which should fall quickly to zero as the optimal solution is neared.

§6 Using OSL for quadratic programming

OSL can apply a generalization of the simplex method (as described in Section §4) to minimize a convex quadratic function, or maximize a concave quadratic function, subject to linear constraints. We refer to these problems as *quadratic programs*, or QPs.

Mathematically, a convex quadratic objective is given by any function that has the form $c^T x + x^T Q x$, where Q is a positive semi-definite matrix. For practical purposes, you can think of a quadratic term as being any expression that involves one linear expression multiplying exactly one other linear expression. An objective function is quadratic if it is a sum of quadratic terms, possibly together with linear terms as would be found in any LP. Thus the simplest convex quadratic objective is a sum of squares:

```
set T;
var X {T};
minimize convex1: sum {j in T} X[j]^2;
```

More broadly, a convex quadratic function may be any sum of squares of linear functions, weighted by any nonnegative values, and added to any linear function:

```
set S;
param w {S} >= 0;

set T;
param c {T};
param q {S,T};

var X {T};
minimize convex2: sum {j in T} c[j] * X[j] +
   sum {i in S} w[i] * (sum {j in T} q[i,j] * X[j]) ^ 2;
```

These are only the most common special cases, however. In general, any quadratic function is convex if and only if the sum of the quadratic terms is ≥ 0 for any values (feasible or not) of the variables. A quadratic function is concave if and only if its negative is convex.

AMPL automatically detects when you have specified a quadratic objective subject to linear constraints, and passes this information to OSL. A minimization is sent directly, while a maximization is converted to a minimization by sending OSL the negative of the objective. Thus all messages from OSL refer to the minimization of a convex function, even if you are really maximizing a concave one.

By forming the matrix Q and applying certain tests, OSL checks for the convexity of the quadratic objective. If the objective fails one of OSL's tests, AMPL alerts you with an error message such as the following:

```
Abandoning QP algorithm:
the quadratic objective function is not convex because
the Q matrix has a negative diagonal element for variable 2.

Abandoning QP algorithm:
the quadratic objective function is not convex because
a principal minor of the Q matrix has determinant < 0.
```

Normally OSL refuses to solve your problem in these cases, but you can insist that it continue by use of the `convex_qp` directive described below.

If you expect OSL to be solving a quadratic program but AMPL reports

```
Sorry, osl can't handle nonlinearities.
```

then you have inadvertently included either a non-quadratic function in the objective, or some nonlinear function in the constraints. Keep in mind that AMPL treats ''almost linear" functions like `abs` and `min` as nonlinear for this purpose.

If your objective is quadratic and all your constraints are linear, but AMPL reports

```
Sorry, osl can't handle integer QPs.
```

then you must have included some variables declared `integer` or `binary` in your model. Although OSL can handle quadratic objectives and can handle integer variables (Section §7), it cannot deal with both together.

There are a few directives, described below, that have special meaning for OSL's quadratic programming algorithm. OSL also recognizes `logfreq` and `maxiter` when solving quadratic programs, with the same meanings that they have for the linear programming case (Section §4).

`majorits=` *i* **(default 30)**
`dccutoff=` *r* **(default 0.0)**

To get a good starting point for its quadratic programming algorithm, OSL employs a ''decomposition crash''. This procedure iterates between linear *subproblems* that give lower bounds on the objective value, and one-constraint quadratic *master problems* that give upper bounds on the objective. One round of solving and updating a subproblem and a master problem is regarded as a *major iteration* of the decomposition crash.

The decomposition crash procedure continues until either *i* major iterations have been performed, or the gap between the best lower and upper bounds is less than *r*. The amount of memory required by the procedure grows in proportion to i^2, so *i* should not be set too large. A message such as

```
At least 14037 additional entries in DSPACE are needed.
```

may indicate that you must either supply more memory by use of the `dspace` directive (Section §3) or must reduce the value of *i*.

`convex_qp=` *i* **(default 1)**

At its default value of 1, this directive causes OSL to reject any minimization of a quadratic function that is not convex. Since objectives to be maximized are negated before being sent to OSL, as explained above, this setting also effectively causes OSL to reject any maximization of a quadratic function that is not concave.

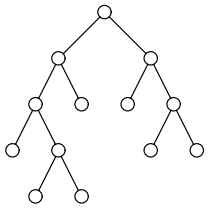
Setting *i* to 0 instructs OSL to apply its quadratic programming algorithm regardless of the convexity of the objective. This option may be useful if you know that the objective is convex within the feasible region defined by the constraints, or if you have any other reason to believe that the quadratic programming algorithm will return a local minimum of interest. Care must be taken in interpreting the results in this case, however, since in general the ''optimal'' solution found by the algorithm need not be minimal in any sense. The algorithm may also terminate with the message,

```
Abandoning QP algorithm because of negative curvature (nonconvexity).
```

indicating that it did not locate a minimum, yet at some iteration was unable to generate a step to improve the objective.

§7 Using OSL for integer programming

For problems that contain integer variables, OSL uses a branch-and-bound approach. The optimizing algorithm maintains a hierarchy of related linear programming *subproblems*, referred to as the *search tree*, and usually visualized as branching downward like this:



There is a subproblem at each *node* of the tree, represented by a circle in the diagram.

The algorithm starts with just a top (or *root*) node, whose associated subproblem is the continuous *relaxation* of the integer program — the LP that results when all integrality restrictions are dropped. If this relaxation happened to have an integer solution, then it would provide an optimal solution to the integer program. Normally the optimum for the relaxation has some fractional-valued integer variables, however. A fractional variable is then chosen for *branching*, and two new subproblems are generated, each with more restrictive bounds on the branching variable; for example, if the branching variable is binary (or 0-1), one subproblem will have the variable fixed at zero, and the other will have it fixed at one. In the search tree, the two new subproblems are represented by two new nodes connected to the root. Most likely each of these subproblems also has fractional-valued integer variables, in which case the branching process must be repeated; successive branchings produce the sort of tree structure shown in our diagram above.

If there are more than a small number integer variables, the branching process has the potential to create more nodes than any computer can hold. There are two key circumstances, however, in which branching below a particular node can be discontinued:

- The node's subproblem has no fractional-valued integer variables. It thus provides a feasible solution to the original integer program. If this solution yields a better objective value than any other feasible solution found so far, it becomes the *incumbent*, and is saved for future comparison.
- The node's subproblem has no feasible solution, or has an optimum that is worse than a certain *cutoff* value. Since any subproblems under this node would be more restricted, they would also either be infeasible or have an optimum value that is no better. Thus none of these subproblems need be considered.

In these cases the node is said to be *fathomed.* Because subproblems become more restricted with each branching, the likelihood of fathoming a node becomes greater as the algorithm gets deeper into the tree. So long as nodes are not created by branching too much faster than they are inactivated by fathoming, the tree can be kept to a reasonable size.

Eventually fathoming overtakes branching, and the number of *active* nodes (from which branching is still possible) begins to shrink. When no active nodes are left, OSL is finished, and reports the final incumbent solution back to AMPL. If the cutoff value has been taken as the objective value of the current incumbent — OSL's default strategy — then the reported solution is declared optimal. Other cutoff options, described below, cannot provide a provably optimal solution, but may allow the algorithm to finish much faster.

Because a single integer program generates many LP subproblems, even small instances can require large amounts of computing time and storage space. In contrast to solving linear programming problems using OSL, where little user intervention is required to obtain optimal results, you will typically have to set some of the following directives to get satisfactory results on integer programs.

Directives for setting up a problem

Any linear model that contains variables declared `integer` or `binary` is treated by AMPL as an integer programming model. When OSL is chosen as the solver, any instance of such a model is

by default set up for solution by OSL's branch-and-bound routines, overriding any of the algorithm selection directives defined in Sections §4 and §5. (Quadratic programs as defined in Section §6 cannot be solved by OSL if they also have integer variables.)

The size of the integer program presented to OSL's branch-and-bound procedure may be reduced by AMPL's presolve phase (described in Section 10.2 of the AMPL book), or by the OSL presolve routines that are controlled by the `simplify` directive (described in Section §3 above). AMPL's presolve takes integrality into account when it attempts to tighten the bounds on variables, while OSL's presolve works the same regardless of integrality restrictions. OSL also has an elaborate collection of presolve routines that are designed specially for 0–1 variables; they are described separately in the next subsection.

`relax`

This directive instructs OSL to solve only the continuous relaxation of any integer program that it receives, ignoring all integrality restrictions.

You can achieve a similar effect by changing the AMPL option `relax_integrality` from its default of 0 to 1, to request that AMPL ignore the keywords `integer` and `binary` in all your model's declarations of variables. This alternative may result in a somewhat lower minimum or higher maximum being reported, however, because it causes AMPL's presolve phase to operate on the continuous linear relaxation rather than on the original integer problem. As a simple example, if x is an integer variable then the constraint $2x \leq 15$ will normally be tightened by AMPL's presolve to $2x \leq 14$, even if OSL's `relax` directive is specified; but setting the AMPL option `relax_integrality` to 1 suppresses this tightening.

`sos2=` *i* **`(default 1)`**

Piecewise-linear programs that do not meet the criteria for transformation to a linear program — as described in Section §1 above — are transformed automatically to integer programs for solution by the branch-and-bound routines.

When this directive is left at its default value of 1, the transformation employs ''special ordered sets of type 2'' that help OSL search for the solution in an efficient way. If the value is changed to zero, the transformation instead creates a somewhat larger integer program that does not use special ordered sets.

Directives for controlling the 0-1 preprocessing heuristics

If your integer program contains variables that may only take the values 0 or 1 — such as variables declared `binary` in the AMPL model — then OSL attempts to simplify and tighten the constraints on these variables before it begins the branch-and-bound process. It also optionally performs a simpler form of the same analysis at certain nodes below the root; these are called *supernodes*, because the analysis is comparable to analyzing many nodes of the branch-and-bound tree at once.

Each pass of this preprocessing phase employs a collection of heuristic procedures as follows:

- *Bound analysis*: Perform an analysis that may tighten the bounds on some constraint sums (without cutting off any feasible solutions). If any constraint's lower bound is shown to lie above its upper bound, declare the subproblem at the current node to be infeasible.
- *Solution search*: Try to find a feasible solution quickly, using a heuristic approach to set each 0-1 variable to either 0 or 1.
- *Probing*: Tentatively set each 0-1 variable to 0 and then to 1, and determine the consequences for all other variables. If setting the variable both to 0 and to 1 gives rise to an unsatisfiable condition, then the subproblem at the current node is infeasible. If only setting the variable to 0 (or to 1) gives rise to an unsatisfiable condition, then the variable may be fixed at 1 (or 0). The effect of this processing is similar to branching in branch-and-bound, but it uses a logical analysis that does not require solving a series of linear programs.

- *Coefficient strengthening*: For each variable that is not fixed by probing, determine whether setting it to 0 or to 1 causes any constraint in the matrix to become redundant. If so, the coefficient for that variable in that constraint may be modified so that the constraint remains the same if the variable is set the other way, but is stronger when the variable lies between 0 and 1 — in the sense that the LP relaxation gives an objective value closer to the integer optimum.
- *Cutting*: In the course of probing, OSL determines whether setting a variable to 0 or to 1 implies that other variables are fixed to their bounds. The resulting ''implication list'' is used to add new constraints, called *cuts*; for example, if setting x to 1 implies that y must be zero, the constraint $x + y \leq 1$ can be added. OSL looks for cuts that are violated by the current subproblem, since these will strengthen the continuous relaxation without changing the optimal value of the integer problem.

Passes are repeated as long as variables continue to be fixed, or the objective value of the relaxation continues to be improved. Then, except when the current node is the root, an additional heuristic fixes one or more 0-1 variables at 0 or 1 — essentially extending the branch-and-bound tree — and further passes of the above procedures are carried out. The cycle is repeated until a feasible integer solution is obtained or a subproblem is determined to be infeasible.

The following directives control how and when the 0-1 preprocessing heuristics are applied, and are significant only when OSL has been passed a problem that contains 0-1 variables.

`pretype=` *i* **`(default 1)`**

When this directive is set to 2, the 0-1 preprocessing heuristics are applied both initially at the root node, and at subsequent supernodes. The default setting of 1 suppresses the supernode option, so that 0-1 preprocessing occurs only at the root node. A setting of 0 suppresses all 0-1 preprocessing.

The preprocessing heuristics tend to require time on the order of k^2 or k^3, where k is the number of nonzeros in the constraint matrix, while the branch-and-bound procedures tend to require time on the order of 2^n, where n is the number of integer variables. Thus on some problems, particularly small ones, the time taken by preprocessing may exceed the time saved, and a setting of 0 is most appropriate. For more difficult applications, some experimentation may be necessary to determine whether a setting of 1 or 2 is most effective.

`prestrat=` *i* **`(default 1)`**

This directive controls two strategies for the 0-1 preprocessing heuristics. Setting *i* to 0 selects neither strategy; setting *i* to 1 or 2 selects strategy 1 or 2 as described below; and setting *i* to 3 selects both strategies.

When strategy 1 is off, probing is performed on all 0-1 variables, and is used to separately analyze the consequences of fixing each of these variables to 0 and to 1. Selecting strategy 1 causes probing to be performed only on 0-1 variables that take the value 0 or 1 in the continuous relaxation at the current node; and probing only determines the consequences of fixing each of these variables at its current value.

When strategy 2 is off, OSL incorporates two somewhat different solution search heuristics: one to search for the first feasible integer solution, and one to search for a better integer solution when an incumbent integer solution has already been found. Selecting strategy 2 causes OSL to use only the latter heuristic. This may be useful in situations when you know the objective value of some feasible solution — perhaps found by some earlier run — and you have specified this value through use of the directive `bbcutoff` (described below).

It is difficult to predict which of these strategies will improve performance on a particular problem. The default tends to perform best overall, but there are many problems for which a different choice can provide significantly improved performance. You should keep in mind, however, that there may also be choices that degrade performance. If you are going to run a large, difficult model repeatedly, experimentation with the setting of this directive may be worthwhile.

cutmult= *r* **(default max{0.25, 100/***m***},**
where *m* **= number of constraints)**

This directive governs the relative number of additional constraints (or cuts) that the 0-1 preprocessing heuristics may add. The maximum number added is equal to *r* times the original number of constraints in the problem.

prepassmax= i_1 **(default 1)**
prepassbranch= i_2 **(default ∞)**
prepassfix= i_3 **(default 1)**
prepassimprove= *r* **(default 0.0)**

These directives govern the number of passes made by the 0-1 preprocessing heuristics. If *r* is left at its default value of 0.0, OSL sets it automatically by use of a heuristic formula.

The preprocessing at the root node (before the branch-and-bound procedure begins) is terminated after i_1 passes, or when the latest pass has fixed fewer than i_3 variables *and* has improved the objective value of the relaxation by less than *r*.

The preprocessing at a supernode is terminated when all of the following conditions are true: at least i_2 implicit branches have been taken, the latest pass has fixed fewer than i_3 variables, and the latest pass has improved the objective value of the subproblem by less than *r*. When i_2 is left at its default value of ∞, preprocessing continues until either a feasible integer solution is found, or a subproblem is determined to have no feasible solution.

Directives for controlling the branch-and-bound search

At each step of the branch-and-bound procedure, OSL extends the tree by choosing a node to branch from, and a fractional variable (from the solution to that node's subproblem) to branch on. The following directives guide the exploration of the tree by influencing OSL's node and branch selection strategies.

It is difficult to predict which node-choice and branch-choice strategies will improve performance on a particular problem. The defaults indicated below tend to perform well overall, but there are many problems for which a different setting can provide significantly improved performance. You should keep in mind, however, that there are also many settings that can degrade performance. If you are going to run a large, difficult model repeatedly, experimentation with these directives may be worthwhile.

fracweight= *r* **(default 1.0)**

For each integer variable, OSL maintains a *pseudocost*: an estimate of the rate at which the objective value will be degraded as that variable is pushed from its optimal fractional value (in the subproblem at some node) toward an integer value. Specifically, the *up-pseudocost* is the estimated rate of degradation as the variable is pushed up toward the nearest higher integer value, and the *down-pseudocost* is the estimated rate of degradation as the variable is pushed down toward the nearest lower integer value.

After creating each new node and solving the associated subproblem, OSL looks for integer variables that still have fractional values. For each fractional variable *x*, the degradation in the objective value that will be caused by making it integral is estimated as

$$r + \min(f_1, f_2),$$

where

f_1 = (ceil(x) – x) (up-pseudocost),
f_2 = (x – floor(x)) (down-pseudocost).

These estimates are summed to give a total *degradation* for the new node: an estimate of how much worse the objective will get before an integral solution is reached by branching from the

node. The estimated degradations influence subsequent node and branch selection as explained below.

When *r* is left at its default value of 1.0, the computed total degradation reflects both the number of infeasible (fractional) variables and the degree of infeasibility of each variable. A larger value of *r* gives more weight to the number of infeasibilities, and tends to give better performance for problems that have a high proportion of 0-1 variables. A smaller nonnegative value of *r* gives more weight to each variable's degree of infeasibility and its pseudocosts.

degscale= r_1 **(default 1.0)**
target= r_2 **(default described below)**

Until the first feasible integer solution is found, OSL always branches from the most recently created available node. This ''depth-first search'' strategy is designed to find an initial integer solution as quickly as possible.

After a feasible integer solution has been found, OSL chooses a node at each step as follows. For a minimization, two values are associated with every node:

- a *current* objective value equal to the optimum of the node's (relaxed) subproblem;
- an *estimated* objective value equal to the current objective value plus r_1 times the node's total degradation (as defined under `fracweight` above).

The chosen node is one that minimizes the estimated objective value, over all active nodes that have a current objective value less than r_2. If no node has a current objective value less than r_2, then the minimum is taken over all active nodes.

The analogous rule is used for a maximization. The estimated objective value is defined with ''minus'' instead of ''plus'', and the chosen node is one that maximizes the estimated objective value, over all available nodes that have a current objective value greater than r_2.

A small value of r_1 biases OSL's choice toward nodes that have a good current objective value, but that may be far from a feasible integer solution. A large value of r_1 biases the choice toward nodes that are likely to be near a feasible integer solution. If r_1 is specified as zero then OSL automatically resets it after each step so that the estimated objective value at the most recently created node comes out equal to the target value r_2; this can be a good way of generating reasonable values of r_1 automatically.

Generally r_2 should be set to an objective value that is known or expected to be attainable by some feasible integer solution. If no value is specified, r_2 is set to 1.05 times the objective value of the continuous relaxation at the root node.

branch= *i* **(default 0)**

This directive controls the selection of four strategies pertaining to branching and pseudocosts, described below. The strategies are numbered by powers of two — 1, 2, 4, 8 — and *i* must be set to the sum of the numbers that correspond to the strategies you want. For example, to select only the strategies numbered 2 and 8, specify *i* = 2+8 = 10.

1: When this strategy is off, OSL chooses to branch on a variable whose smaller pseudocost is largest. As a result, the branch-and-bound tree tends to be extended from a node where both branches cause significant degradation in the objective function, probably allowing the tree to be pruned earlier. Selecting this strategy causes OSL to instead take the branch opposite the maximum pseudocost. This strategy tends to force the most nearly integer fractional variables to integers earlier, and may be useful when any integer solution is desired, even if not optimal. The search tree tends to grow much larger, but if the search is successful and an adequate integer solution is found, then most of the tree will never have to be explored.

2: When this strategy is off, OSL computes the pseudocosts only once for each variable. Selecting this strategy causes OSL to update the pseudocosts after each branch.

4: When this strategy is off, OSL computes the pseudocosts only for variables that take fractional values at the root node of the branch-and-bound process. Selecting this strategy causes OSL to compute a pseudocost for a variable that is integral initially but that changes to a fractional value

in the continuous relaxation at some descendant node. (The pseudocost is computed only the first time that the variable is found to be fractional.)

8: When this strategy is off, OSL computes the pseudocosts only for variables that lie at fractional values in optimal solutions to subproblems. Selecting this strategy causes OSL to also compute pseudocosts for variables that lie at integer values.

Directives for limiting the search

In dealing with a difficult integer program, you may need to settle for a ''good'' solution rather than a provably optimal one. The following directives let you adjust certain fathoming criteria for OSL's branch-and-bound search. By weakening a fathoming criterion, you may be able to reduce the number of nodes that OSL must process in order to get acceptably close to an optimal solution.

bbcutoff=r_1 **(default** ∞**)**
bbimprove=r_2 **(default 0.00001)**

These directives govern OSL's branch-and-bound cutoff value, c. A node is fathomed if its current objective value is less than c (for maximization problems) or greater than $-c$ (for minimization problems).

Initially, c is set to r_1. Each time that a new incumbent integer solution is found, c is reset to the objective value of that solution minus r_2.

The default values of these directives cause the branch-and-bound procedure to behave in a ''normal'' way. Initially OSL searches for any feasible integer solution at all, and does not fathom any node where the subproblem is feasible. After an incumbent solution has been found, OSL fathoms any node where the subproblem's objective value does not improve on the incumbent's objective value by at least 10^{-5}.

If r_1 is set to a finite value, then OSL will only consider nodes that have a current value $\leq r_1$ (for a minimization) or $\geq -r_1$ (for a maximization). As a result, the branch-and-bound procedure may search fewer nodes and may establish an optimum sooner; but too low a value of r_1 may result in no integer solution being found. When minimizing, it makes sense to set r_1 to the objective value of any known feasible integer solution, or to an upper bound on the optimum objective value; similarly when maximizing, it makes sense to set $-r_1$ to the objective value of any known feasible integer solution, or to a lower bound on the optimum.

If r_2 is set to a larger value, then the branch-and-bound procedure is forced to ignore integer solutions that are not at least r_2 better than the one found so far. Again the result tends to be fewer nodes searched, and quicker termination; but the true integer optimum may be missed if its objective value is within r_2 of the best integer objective found. It makes sense to set r_2 to the smallest difference between two objective values that you would consider to be significant for your purposes.

If only integer variables figure in the objective, and if all such variables have integral coefficients, then the objective value is always an integer. In such a case you may be able to accelerate the branch-and- bound procedure by setting r_2 to one minus a small tolerance — say, to 0.9999 — without any risk that the true optimum will be missed.

tolint= r **(default 1.0e-6)**

In the optimal solution to a subproblem, a variable is considered to have an integral value if it lies within r of an integer. For some applications, specifying a larger r may reduce the time spent in the branch-and-bound procedure, while allowing OSL to return a solution that may be rounded (by use of AMPL commands) to an acceptable result. (Various features useful in rounding the actual or displayed values of the variables are discussed in Sections 10.5 and 10.8 of the AMPL book.) For other applications, particularly those using 0-1 integer variables, the default value of r is most appropriate.

Directives for stopping the search

A partial run of the branch-and-bound search is often sufficient to determine the impact of different settings of the above directives. While you are experimenting, consider using one of the following directives to set a stopping criterion in advance. In each case, the best solution found so far is returned to AMPL.

maxiter= i_1 **(default** ∞**)**
iter_inc= i_2 **(default 2000)**

The branch-and-bound algorithm is stopped after a total of i_1 simplex iterations have been used in solving all the subproblems. OSL then uses up to i_2 additional simplex iterations to recover the best integer solution that has been found so far.

A warning is printed if i_1 is too small to permit any integer feasible solution to be found, or if i_2 is too small to permit the best integer solution to be recovered.

maxnodes= i **(default** ∞**)**

The search is terminated after i nodes of the branch-and-bound search tree have been processed.

maxsols= i **(default** ∞**)**

The search is terminated after i feasible solutions satisfying the integrality requirements have been found.

mingap= r **(default 0.0)**

The search is terminated when the ''gap'' between OSL's lower and upper bounds on the true objective value falls to less than r. After such a termination, the solution returned by OSL is guaranteed to achieve an objective value that is within r of the best possible.

For a minimization, the upper bound is given by the objective value of the incumbent integer solution, while the lower bound is derived by OSL from the objective values of certain fractional solutions at subproblems. For a maximization, the situation is the same except with the roles of ''lower'' and ''upper'' exchanged. The `bbdisplay` and `bbdispfreq` directives (described below) permit you to monitor the gap's reduction through the course of a long run.

Directives for managing memory

OSL's memory requirement for solving linear subproblems is about the same as its requirement for linear programs discussed in Section §4. For the branch-and-bound procedure, however, each active node of the tree requires additional memory. The total memory that OSL needs to prove optimality for an integer program can thus be much larger and less predictable than for a linear program of comparable size.

The `dspace` directive described in Section §3 above lets you increase the memory available to OSL's branch-and-bound routines. The following directives permit you to trade off memory space against disk space, by directing where certain node information will be written.

bb_mfile=i_1 **(default 1)**
bb_bfile=i_2 **(default 1)**

OSL maintains two space-consuming pieces of information associated with each node. The *matrix block* specifies how integer variables have been bounded or fixed in the current subproblem, and the *basis map* records the subproblem's optimal basis.

By setting i_1 or i_2 to 0, you direct OSL to keep in memory all matrix blocks or basis maps, respectively. When both are set to 0, the size of OSL's branch-and-bound tree is limited by the size of the OSL's memory region, which you may enlarge by use of the `dspace` directive. An error message such as

```
There is not enough storage to extend buffers
```

indicates that you need a larger `dspace` setting to hold the search tree.

If you leave these directives at their default values of 1, then only a few recently used matrix blocks and basis maps are stored in buffers in memory. Most are written instead to temporary files `fort.4` and `fort.3`, respectively, in the directory that was current when AMPL was invoked. The size of OSL's branch-and-bound tree is thus limited by the amount of free disk space in the file system where these temporary files are being written. An error message such as

```
/usr: write failed, file system is full
   There is an I/O error on direct access file 3 at record 2434
```

indicates that you need more disk space to hold the search tree.

The names and locations of `fort.4` and `fort.3` are fixed by OSL. Thus both i_1 and i_2 should be set to 0 if you expect several AMPL processes to be run at the same time from the same directory, and there is any chance that two or more of these processes may attempt to apply OSL to an integer program at the same time. An error message such as

```
OSL data was overwritten; this was detected by subroutine EKKMSLV
```

indicates that two or more processes have been trying to use the same `fort.3` or `fort.4` at the same time.

Directives for controlling output

Unless you are solving a very easy integer program, you may want to use the following directives to monitor the progress of the branch-and-bound procedure. You may also request more complete diagnostic information by increasing the setting of the `outlev` directive described previously in Section §3.

bbdisplay=i_1 **(default 0)**
bbdispfreq=i_2 **(default 9999999)**

The default of $i_1 = 0$ produces a minimal few lines of output from OSL, summarizing the results of the run.

When $i_1 = 1$, a ''log line'' is printed for each new optimal solution found:

```
ampl: model multmip3.mod; data multmip3.dat;

ampl: option solver osl;
ampl: option osl_options 'bbdisplay 1';

ampl: solve;
OSL 1.2:
bbdisplay 1
 The dual algorithm has been chosen

Searched  Intsols  Best            Bound           Gap
      24        1  244025          226049          1.8e+04
      58        2  242025          227823          1.4e+04
      96        3  237125          228412.83       8.7e+03
     125        4  235625          234494          1.1e+03
OSL 1.2: optimal solution; objective 235625
838 simplex iterations; 132 branch-and-bound nodes
4 feasible integer solutions found
ampl:
```

The first two columns represent the number of nodes searched (branched on) so far, and the number of integer feasible solutions found so far. Next are the objective value of the incumbent solution — the best integer feasible solution found so far — and a bound on the objective, derived by OSL from the fractional subproblem solutions. These are an upper and a lower bound, respectively, on the optimum for minimization problems, and a lower and upper bound, respectively, on the optimum for maximization problems. The fifth column gives the difference between the two bounds; the incumbent's objective value and the true minimum or maximum cannot differ by more than this ''gap''.

When $i_1 = 2$, an additional log line is printed every i_2 nodes searched. These lines permit you to monitor the reduction in the gap at regular intervals.

When the branch-and-bound procedure is applied to a difficult integer program, it may search thousands of nodes without finding any improvement in the incumbent solution. Moreover, whereas the simplex and barrier methods for continuous linear programming usually find an optimal solution only at or near the last iteration, branch-and-bound may process a huge number of nodes even after the optimum integer solution has been found; in such cases, the additional nodes must be searched in order to prove that there is no better solution. Often for practical purposes it is better to accept the best solution found at some point, particularly when the gap is small, than to insist on a provably optimal solution that may require far more computational effort. Several of the directives described previously can be used to stop the search, though some experimentation may be necessary to determine good settings for a particular application.